VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Abstraktní interpretace

Abstract Interpretation

2015 Jakub Kermaschek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Jakub Kermaschek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Abstraktní interpretace
Abstract Interpretation

Zásady pro vypracování:

Abstraktní interpretace je jednou ze základních technik používaných pro statickou analýzu programů. Představuje určité zobecnění přístupů používaných pro analýzu datového toku (data-flow analysis), kde se řeší typicky problémy jako jsou například reaching definitions, live variables, available expressions, very busy expressions apod. Abstraktní interpretace zahrnuje jako speciální případy všechny tyto typy analýz a kromě toho celou řadu dalších jiných typů analýz, například analýzu intervalů.
Cílem práce je implementovat základ, který je společný pro všechny algoritmy využívající princip abstraktní interpretace, a implementovat několik jednodušších typů analýz.

1. Nastudujte příslušnou problematiku.
2. Implementuje obecné algoritmy pro abstraktní interpretaci a jako příklad několik typů konkrétních analýz. Systém pro abstraktní interpretaci navrhněte tak, aby se do něj daly další typy analýz v budoucnu snadno přidávat.
3. Otestujte implementaci na vhodně zvolených testovacích programech.

Seznam doporučené odborné literatury:

[1] Flemming Nielson, Hanne Riis Nielson, Chris Hankin: "Principles of Program Analysis". Springer, 1999.
[2] Patrick Cousot, Rathia Cousot: "Static determination of dynamic properties of programs". 2nd International Symposium on Programming, Paris, pp. 106-130, 1976.
[3] Paris Cousot, Rathia Cousot: "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, pp. 238-252, ACM Press, 1977.

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **doc. Ing. Zdeněk Sawa, Ph.D.**

Datum zadání: 01.09.2014

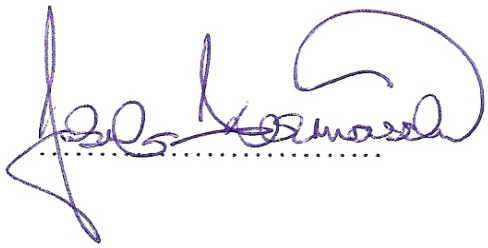
Datum odevzdání: 07.05.2015

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 7. května 2015



Rád bych na tomto místě poděkoval svému vedoucímu bakalářské práce doc. Ing. Zdeňku Sawovi, Ph.D. za odborné vedení a rady při její tvorbě.

Abstrakt

Cílem bakalářské práce je vytvořit framework, ve kterém je implementována abstraktní interpretace. Abstraktní interpretace spadá do oblasti statické analýzy programů. Programy, které jsou analyzovány jsou napsané v nově vytvořeném jazyce nazvaném KerLang. Pro příklad jsou zde uvedeny konkrétní typy abstraktních interpretací. V práci je nejprve uvedeno vytvoření samotného programovacího jazyka, s kterým tento framework pracuje, včetně vysvětlení syntaxe, gramatiky a příkladů programů. Kromě těchto základních příkladů se zde věnuji průběhu samotné syntaktické analýzy, vytvoření abstraktního syntaktického stromu a nakonec vytvoření grafu řídícího toku. Další část práce se soustředí na postup vytvoření frameworku abstraktních interpretací a následných rozborů konkrétních druhů analýz. Všechny tyto části také detailně popisují průběh implementace, samotného rozvržení a architektury kódu.

Klíčová slova: Statická analýza programů, abstraktní interpretace, živé proměnné, analýza intervalů, syntaktická analýza, .NET framework

Abstract

The goal of this bachelor thesis is to create a framework, in which these abstract interpretations are implemented. Abstract interpretation is part of static analysis of programmes. Programmes, which are analysed are written in new programming language, which is called KerLang. For an example, there are given specific types of the abstract interpretations. In the thesis there is at first stated the creation of the language itself, with which this framework works, including an explanation of a syntax, a grammar and an examples of a programs. In addition to these elementary examples the given part of the thesis is also devoted to a parsing itself, creation of an abstract syntax tree and finaly creation of a flow control graph. The next section of the thesis is focused on creation of a framework of abstract interpretation and subsequent analysis of specific types of analysis. All these sections also describe a detail of implementation, the actual layout and architecture of the code.

Keywords: Static analysis of programmes, abstract interpretation, live variable, analysis of intervals, syntactic analysis, .NET framework

Seznam použitých zkratek a symbolů

AST	–	Abstract syntax tree/Abstraktní syntaktický strom
BG	–	Context-free grammar/Bezkontextová gramatika
CFG	–	Control flow graph/Graf řídícího toku

Obsah

1 Úvod **4**

2 Programovací jazyk KerLang **5**
2.1 Obecné vlastnosti 5
2.2 Syntaxe 5
2.3 Gramatika 6
2.4 Konkrétní programy 7

3 Implementace parseru pro jazyk KerLang **9**
3.1 Lexikální analýza 10
3.2 Parser 11
3.3 Abstraktní syntaktický strom 12
3.4 Graf řídícího toku 17

4 Abstraktní interpretace **21**
4.1 Svazy 21
4.2 Průchod CFG 22

5 Konkrétní analýzy **25**
5.1 Analýza intervalů 26
5.2 Živost proměnných 31

6 Práce s programem **34**
6.1 Spuštění programu 34
6.2 Přidání analýzy do frameworku 34

7 Závěr **35**

8 Reference **36**

Přílohy **36**

A Použitý software **37**

B Druhy tokenů jazyka KerLang **38**

C Gramatika KerLang **40**

D Tabulka podmínek a intervalových výsledků **44**

E Složitější kód určen k testování programu **45**

F Příloha na CD/DVD **46**

Seznam obrázků

1 AST příkazu: `a = a + b * c` 14
2 AST′ příkazu: `a = a + b * c` 15
3 Konkrétní svaz 22

Seznam výpisů zdrojového kódu

1 Program pro sečtení dvou celých čísel 7
2 Program pro setřízení prvků v poli za pomoci metody bubble sort 8
3 Princip kontroly chyb v AST 13
4 Interface `ILattice` 25
5 Metoda `Call` neměnící hodnotu intervalů 29

1 Úvod

Hlavní náplní této bakalářské práce je především vytvoření frameworku pro abstraktní interpretaci, což je vlastně jedna ze základních technik používáná pro statickou analýzu programů.

Samotná abstraktní interpretace je určité zobecnění přístupů pro analýzu datového toku, kde se řeší problémy jako jsou reaching definitions, live variable, available expressions, very busy expression apod.

Pro samotnou abstraktní interpretaci je potřeba programu, respektive kódu, nad kterým se tyto statické analýzy provádějí. Abych mohl tyto analýzy implementovat a následně předvést jejich použití, musel jsem prvně naimplementovat programovací jazyk, který jsem nazval KerLang. V tomto jazyce, jsou tyto programy napsány. Poté jsem musel také naimplementovat vytvoření abstraktního syntaktického stromu, dále jen AST, ze kterého se vytváří graf řídícího toku, dále jen CFG, který je potřeba pro danou statickou analýzu.

Využití konkrétního typu analýz obecně slouží k optimalizaci kódu. Například analýza available expressions nám může zredukovat náročnost kódu tím, že se nepřepočítává něco co již bylo jednou spočteno a novým přepočtením by jsme se dobrali ke stejnému výsledku. Zatímco třeba analýza živých proměnných nám zase říká, kdy je potřeba si danou hodnotu proměnné držet v paměti a kdy ji můžeme přepsat bez toho aniž by se ovlivnil výsledek programu.

Díky abstraktní interpretaci se programy, respektive jejich kódy, mohou přetransformovat do verzí, které jsou pro člověka sice méně čitelné, ale pro samotný stroj méně náročné. Výsledkem abstraktní interpretace je zanalyzovaný kód. Tuto analýzu pak dále může využít překladač a podle ní měnit strukturu kódu. Výsledek programu podle nového kódu, je totožný s výsledkem původní verze kódu, ovšem tento návrh má nižší nároky na hardware.

Samotný text bakalářské práce se tedy zaobírá tím, jak jsem daný problém řešil. Jak jsem vytvořil daný programovací jazyk a jak jsem naimplemetoval daný framework. Výsledkem bakalářské práce je program, který je uveden jako příloha na disku.

Celá tato programovací část je implementována v jazyce C#, v prostředí Visual Studio 2013.

2 Programovací jazyk KerLang

V této kapitole je obecně popsán jazyk KerLang, jsou zde ukázany jeho vlastnosti, vysvětlena jeho syntaxe a gramatika. Také zde uvádím konkrétní příklady programů, napsané v jazyce KerLang.

KerLang je imperativní programovací jazyk, který syntaxí připomíná jazyk Pascal. Syntaktická analýza a vytvoření CFG, je navrženo tak, aby bylo možné jednoduše přidat další rozšíření jazyka, například přidání datových typů.

2.1 Obecné vlastnosti

Programovací jazyk KerLang je velice jednoduchý jazyk. Program (syntaktická analýza KerLang a framework pro abstraktní interpretaci) je implementován v jazyce C#. Je navrhnut tak, aby jeho syntaxe byla co nejjednodušší, ale aby zároveň obsahoval všechny základní konstrukce, nad kterými má statická analýza smysl.

Konkrétní program napsaný v tomto jazyce, musí obsahovat:

- Hlavní funkci — program

Uživateli dále dovoluje používat:

- Pomocné funkce — function
- Celočíselný datový typ
- Strukturu pole
- Základní aritmetické operace — sčítání, odečítání, násobení a dělení
- Základní booleovské operace — větší, větší rovno, menší, menší rovno, rovná se a nerovná se
- Cykly — for, while a do-while
- Příkazy pro opuštění cyklu, či funkce - break, continue a return
- Podmínky — if, elseif a else

2.2 Syntaxe

Syntaxe je souborem pravidel, který definuje znaky a jejich kombinace, které se mohou v konkrétním kódu vyskytovat. Tyto kombinace znaků tvoří tokeny[1].

[1] V tomhle případě se tokenem rozumí obecné označení pro aritmetické operátory, znaménka, logické operátory, závorky atp.

Při vytváření syntaxe pro KerLang jsem se nechal inspirovat programovacími jazyky Pascal a C. Díky tomu, že klíčová slova se stejným významem jsou v KerLang a výše zmíněných jazycích stejná, je jednodušší napsat v KerLang nějaký program.

Tabulka, popisující všechny druhy tokenů KerLang je popsána v Příloze B[2]

2.3 Gramatika

Gramatika použita v KerLang je bezkontextovou gramatikou, dále jen BG. BG je určena konečnou množinou terminálů, konečnou množinou neterminálů, konečnou množinou přepisovacích pravidel a počátečním neterminálem. Tato čtveřice BG je popsána v Příloze C.

2.3.1 Gramatika KerLang

Samotná gramatika KerLang neurčuje všechny *korektní* výskyty tokenů v kódu. Naopak připouští výskyty, které nejsou správné, čili jsou pro KerLang nekompilovatelné. Tyto situace se ošetří později v AST, viz. Podkapitola 3.3.

Důvodem této volnosti je také jednoduchost dané gramatiky, pokud by jsme se rozhodli napsat gramatiku, která přesně určuje všechny výskyty, byla by tato gramatika mnohem složitější, za předpokladu, že by vůbec nějaká BG pro popsání všech pravidel existovala. Z praktického hlediska je tedy vhodné neřešit gramatikou všechny přípustné výskyty, ale naopak vyřešit jejich většinu v obecném hledisku a specifická pravidla dále odladit v následujících krocích kompilace, kde už se vlastně jedná jenom o jednoduché podmínky.

2.3.2 Vlastnosti gramatiky KerLang

Z gramatiky je vidět, že její cíl je nadefinovat předpisy pro běžné charakteristiky v programovacích jazycích, jako jsou například cykly, podmínky, funkce, procedury, hlavní program, inicializace proměnných a polí, či nastavit obecnou aritmetiku. Celkově jde o to vytvořit obecný předpis pro programovací jazyk.

Za povšimnutí zde ale stojí neterminál EXPRESSION, který určuje aritmetické, či booleovské výrazy. V gramatice je třeba vyřešit priority operací, nebo také unární a binární operace, které jsou zastoupeny stejným znakem, jako je například znaménko −.

Gramatika zde v průběhu parsování postupně prochází operace od nejvyších priorit až po ty nejnižší a postupně se vytváří zprava doleva. Toto skládání je velice důležité, při vytváření jakékoliv matematické operace nesmíme totiž zapomínat na asociativitu operátorů. Co se týče například sčítání, ničemu by nevadilo, zda skládáme danou matematickou operaci zprava doleva, či zleva doprava, ovšem tato vlastnost například u odčítání, či dělení chybí. Proto je třeba myslet na tyhle matematické vlastnosti už zde, při vytváření samotné gramatiky.

[2]V tabulce tokenů můžeme vidět sloupec "název tokenu," tento sloupec není pro tuhle kapitolu důležitý, ovšem v následující kapitole potřeba bude, z důvodu přehlednosti jsem namísto dvou skoro totožných tabulek vytvořil pouze jednu obsahující jak token samotný, tak i jeho reprezentaci v KerLang.

2.4 Konkrétní programy

Pro ukázku jak vypadá typický program v jazyce KerLang zde uvádím několik příkladů:

```
//Tento program slouzi ke scitani dvou cisel.
function int sum(int a, int b)
{
  return (a + b);
}

program
{
  int a, b;
  write("Zadejte dve cisla pro soucet:");
  read(a, b);
  int soucet = call sum(a, b);

  write("Soucet je: ", soucet);
}
```

Výpis 1: Program pro sečtení dvou celých čísel

```
//Tento program setřídí pole za pomocí třízení bubble sort.
function void bubbleSort(int a[], int length)
{
  for (int i = 0; i < (length - 2); i++)
  {
    for (int j = 0; j < (length - (i - 2)); j++)
    {
      if (a[j] < a[j+1])
      {
        int tmp = a[j];
        a[j] = a[j+1];
        a = tmp;
      }
    }
  }

  for (int i = 0; i < (length - 1); i++)
  {
    write(a[i]);
  }

  return;
}

program
{
  int length = 10;
  int a[10] = {5, 8, 4 + 5 * 2, 2, 10, 15, 18, 0, -5, 1};

  write("Nesetrizene pole: ");
  for (int i = 0; i < length; i++)
  {
    write(a[i]);
  }

  write("Setrizene pole: ");
  call bubbleSort(a, length);
}
```

Výpis 2: Program pro setřízení prvků v poli za pomoci metody bubble sort

3 Implementace parseru pro jazyk KerLang

V této kapitole je popsána implementace parseru pro jazyk KerLang.

Implementace se skládá ze tří hlavních modulů a dvou pomocných, nicméně stejně důležitých modulů.

Hlavní moduly:

1. **`Scanner`**: Tento modul má na starost znak po znaku, směrem od začátku textu až po jeho konec[3], procházet zdrojový kód jazyku KerLang a nacházet v něm tokeny.
2. **`Parser` a `AST`**: Parser má na starost z jednotolivých tokenů vytvořit AST. Jak daný AST vytvoří, závisí na gramatice jazyka, která je popsána v Kapitole 2.3.
3. **`CFG`**: Poslední modul programu, který slouží k vytvoření CFG za pomocí AST získáného z parsování.

Pomocné části:

1. **`Program`**: Programem je myšlena centrální část kódu, která postupně volá všechny třídy a krok po kroku vytváří ze vstupního souboru jeho CFG.
2. **Transformace `AST` do `AST'`**[4]: Tato transformace je velice důležitou částí kódu. Při vytvoření prvního AST z parsování je třeba tento strom ještě upravit a zjednodušit. Také zde odchytíme všechny nepřípustné vstupy, které neumožňuje odchytit gramatika.

Pro spuštění kompilace zdrojového kódu je očekáván soubor "settings.txt".

Tento soubor se nachází v kořenovém adresáři KerLang, ve složce "Settings". Relativní cesta od .exe souboru KerLang vypadá následovně: "..\..\Settings\settings.txt".

V tomto souboru je třeba nastavit klíčové parametry ke spuštění, tyto parametry jsou tři.

1. **Vstupní kod**: Zde se udává cesta k souboru, který obsahuje kód napsán v jazyce KerLang. Tento soubor musí mít příponu ".txt".
2. **Výstup**: Zde je uvedena cesta ke složce, do které se uloží výstup v textových souborech, těmito výstupy jsou AST, CFG a výsledky analýz abstraktních iterpretací pro vstupní kód.
3. **Počet iterací wideningu**: Zde je očekávána celočíselná hodnota, která určuje maximální počet iterací wideningu[5].

[3] Koncem souboru je myšlen výskyt hodnoty EoF - End of File.

[4] Výrazem AST′ je myšlen transformovaný původní AST. Popisuje stejný kód, ale přidává do něj pomocné proměnné atp.

[5] Význam a celé vysvětlení wideningu je uvedeno v Kapitole 4.

Jestliže je vstupní soubor správně nastaven, stačí jen spustit program a vyčkat než se provede kompilace. Pokud vstupní soubor obsahuje nekorektní informace, například místo čísla je načten řetězec, program vyrozumí uživatele o chybě a ukončí se.

3.1 Lexikální analýza

Celkový princip objektu třídy `Scanner`, je vytvářet tokeny na zavolání ze seznamu řetězců typu String, který získává jako parametr konstruktoru. `Scanner` je malou třídou obsahující tři metody.

- `GetNextToken()`

Tato metoda je stěžejní metodou celé třídy a je v ní obsažená veškerá logika vytváření tokenu. Jak je znázorněno v Příloze B, tato metoda vyhledává v řetězci klíčová slova, které pak vrací jako Enum s příslušnou hodnotou.

Tokeny se v implementaci dají rozdělit do tří hlavních kategorií. Kategorie slov, znaků a kategorie řetězců.

Nejjednodušší je kategorie znaků, kde se v jednom velkém switchi zkoumá aktuální znak a pokud je tento záznam jedinečný, rovnou se vrátí.

Existuje zde pár speciálních případů. Například u znaku + máme dvě možnosti o jaký token se jedná. Buď to může být skutečně `plus`, a nebo se může jednat například o autoinkrementaci, v případě ++. Proto musíme načíst následující znak, ovšem pokud tento následující znak + není, je třeba si pamatovat, že při příštím zavolání této metody, nesmíme načítat další znak, ale musíme použít tenhle nevyužitý z předchozího volání.

K tomu nám slouží pomocné třídní proměnné.

Druhá kategorie je kategorií slov. Princip je jednoduchý, pokud klíčové slovo začíná znakem latinské abecedy, načítáme veškeré další znaky dokud to jsou písmena, nebo čísla. Poté co je načtené celé slovo zkoumáme zda se jedná o klíčové slovo (například `for`, `if`, `while`...) a vrátíme token. Pokud se o klíčové slovo nejedná, tak se jedná o proměnnou s tímto názvem. Kromě toho, že vrátíme token identifikující proměnnou musíme si také do pomocné proměnné uložit název této proměnné.

Poslední kategorií je kategorie řetězců, kde načítáme vše co se vyskytuje za speciálním znakem `"` (uvozovky). Stejně jako v předchozím případě, si musíme tento řetězec uložit do pomocné proměnné. Hodnoty těchto proměnných pak budou získávány parserem, a budou vloženy do AST.

Jediným speciálním případem jsou komentáře. Komentáře v KerLang jsou reprezentovány jako `//`. Pokud scanner objeví tyto dvě lomítka za sebou, vrátí informaci o komentáři, ignoruje vše co se za nimi vyskytuje a rovnou se posune na další řádek vstupního kódu.

- `ReadNextChar()`

Jak samotný název napovídá, metodá slouží k načtení dalšího znaku. Inkrementuje index ukazující na aktuální znak v řetězci dokud jeho hodnota není rovna samotné velikosti řetězce.

V tomto případě oznámí, že inkrementace neproběhla a za pomoci této informace je zavolána metoda načítající nový řádek.

- `ReadNextLine()`

Metoda načte do proměnné další řádek vstupního kódu, který jsme získali v konstruktoru. Také resetuje index ukazující na aktuální znak v řetězci a nastaví jeho hodnotu na nulu. Ve speciálním případě, kdy již jsme na konci vstupního kódu místo změny řádku, vrací token informující o konci souboru.

3.2 Parser

Parser je v porovnání se scannerem značně složitější třídou. Veškerá gramatika popsána v Podkapitole 2.3 je zde implementována. Pro každý neterminál existuje metoda, ve které jsou všechny možnosti, které daný neterminál nabízí. Při hlubším pohledu na gramatiku se dá všimnout, že je napsána tak, aby pokud možno žádná možnost, jak může kód vypadat nezačínala stejným terminálem/neterminálem. Je pak totiž jednodušší určit co musí v kódu být a pokud to tam není, jedná se o chybu.

Ovšem tento přístup se nedá využít ve všech případech. V těchto případech musíme kontroly provádět i později.

Co se týče implementace gramatiky jedná se o velice komplikovaný proces. I přesto, že gramatika KerLang je jednoduchá, je obtížné ji přepsat do kódu. Pro zajímavost jenom přepis této gramatiky obsahuje kolem 1500 řádek kódu, kde se všechny metody volají mezi sebou, takže se nedá napsat postupně řádek po řádku. Je třeba si všechny informace udržovat v hlavě a vědět, kde se vrátit po dopsání nějakého jednoduššího neterminálu.

Ovšem gramatická kontrola není hlavní náplní práce parseru. Hlavní úkol této třídy je vytvořit jeden objekt třídy `AST`. Tento proces podrobněji popíšu v Podkapitole 3.3.

Kromě těchto zřejmých metod přepisovacích pravidel, parser obsahuje ještě tři stěžejní metody a neméně důležitou třídní proměnnou.

Úkolem třídní proměnné `readNext` je oznámit parseru, zda si má požádat scanner o následující token. Jak jsem již popisoval v gramatice, existují zde možnosti prázdných slov. Což znamená, že pokud jsme získali token neodpovídající ničemu co máme v přepisovacích pravidlech, ale daný neterminál nabývá možnosti prázdného slova, je možné, že daný token patří jinému neterminálu. Proto si musíme zachovat informaci o tomhle tokenu a nenačítat další při kontrole jiného přepisovacího pravidla.

Dvě metody `CheckNextToken(EnumToken requirement)` a `Error(String error)` slouží k informování uživatele o chybách ve vstupním kódu. První z těchto dvou metod kontroluje, zda-li získaný token odpovídá požadavku (`requirement`). Pokud neodpovídá, oznámí uživateli na kterém řádku je požadovaný jaký token. Druhá z těchto metod přesněji informuje uživatele o chybové hlášce. Nastává to v situacích, kdy je očekáváno více možných terminálů. Způsob získání informace kde se chyba nachází je uložena v objektu třídy `Scanner`. Jelikož načítá tokeny pouze na zavolání parseru, vždy ukazuje na pozici aktuálního tokenu a je tedy možné tuto informaci využít při výpisu chyby.

Poslední metoda `GetNextToken()` spolupracuje se scannerem a požaduje po něm následující token v kódu. Samozřejmě jen v případě je-li proměnná `readNext` nastavená na `true`. Pokud získaný token je komentář, tak rovnou požadujeme další a tento token zahazujeme. V případě tokenu informujícím o konstantě, proměnné, či řetězci požadujeme ještě po scanneru metadata obsahující hodnotu konstanty, název proměnné, či hodnotu řetězce.

Co se týče práce samotného objektu třídy `Parser`, vytváří z většiny terminálů objekty třídy `AST`, které na konci každého přepisovacího pravidla spojuje do jednoho většího stromu. Neterminály, které se zde nevyužívají při vytváření AST jsou například závorky, informace o závorkách slouží pouze gramatice a v AST už nemají žádnou váhu, protože priority vyčteme přímo ze struktury stromu.

Kromě těchto informací také do každého listu stromu ukládá informace o řádku, z kterého daný token vznikl, kvůli budoucím výpisům chyb. Také mu předává všechny metadata, získané z metod třídy `Scanner` o obsahu tokenu.

3.3 Abstraktní syntaktický strom

`AST` je třídou popisující kód a deterministicky určuje strukturu kódu.

Každý vrchol tohoto stromu je také objektem třídy `AST`, díky tomu se dá s vrcholy jednoduše manipulovat a přesouvat je stromem podle potřeby. Důvod těchto přesunů budeme potřebovat v Podkapitole 3.3.2, během transformací AST do nových jednodušších AST.

Každý tento objekt obsahuje kromě informace o tokenu[6] ze kterého vznikl, také další užitečná metadata popisující specifické tokeny, jako je proměnná, konstanta a řetězec. V těchto případech je třeba si udržet také informaci o názvu (v případě proměnné), o hodnotě (v případě konstanty) a také o obsahu (v případě řetězce), ve všech ostatních případech tyto proměnné neobsahují žádná data a nejsou nikdy inicializované.

Metadata využívající všechny listy stromu jsou informace o řádku kódu, ve kterém byly vytvořeny. Tato informace je důležitá pro uživatele, pokud se totiž v kódu nachází nějaká syntaktická chyba, můžeme tak uživateli říct, kde se tato chyba nachází.

Každý list stromu také obsahuje list potomků stejné třídy, čili třídy `AST`. Tento list určuje celou strukturu stromu.

Při generování stromu je potřeba pouze konstruktoru, nastavující hodnotu listu stromu jako takového (název tokenu) a dále metoda spojující více stromů. Touto metodou je metoda `Append(AST child)`, která zastupuje metodu `Add` třídy `List`.

Dalších 18 metod třídy je rozděleno do dvou skupin. Jedna kontroluje strom a snaží se v něm nalézt chyby, které z principu návrhu nemůže odchytit gramatika. Druhá slouží k transformaci stromu.

3.3.1 Kontrola syntaxe v AST

Skupině kontrolující chyby začínají všechny metody předpodnou `CheckFor...`

[6] informace o tokenu je uložena ve výčtovém typu `EnumToken`, hodnota těchto enumů vychází z tabulky v Příloze B, ve sloupci název tokenu.

Stěžejní metodou, která řídí kompletní kontrolu stromu je metoda `CheckForErrors()`. Tato metoda postupně prochází každým listem stromu a postupně kontroluje, jestli se nejedná o listy, které by mohly obsahovat chybu. Pro představu, zde uvádím ukázku Výpisu [7] 3

```
public void CheckForErrors(AST tmp)
{
    foreach (AST tree in tmp.child)
    {
        if (/* Value je aritmeticke porovnavani (vetsi, mensi...) */)
            CheckForMultipleComparisonError(tree);
        else if (/* Value je prirazeni */)
            CheckForComparisonError(tree);
        else if (/* Value je cyklus */)
            CheckForStatementError(tree);
        else if (/* Value je funkce */)
        {
            if (tree.child[2].child.Count > 0)
                CheckForArrayInitializationError(tree.child[2]);
            CheckForContinueBreakError(tree.child[3]);
        }
        else if (/* Value je hlavni program */)
            CheckForContinueBreakError(tree.child[0]);
        else if (/* Value je blok */)
            CheckForUninitializedArrayError(tree);

        CheckForErrors(tree);
    }
}
```

Výpis 3: Princip kontroly chyb v AST

Jak jde vidět z Výpisu 3 metoda postupně projde všechny listy a pokud se jedná o list nějakého určitého typu, zavolá se na něj kontrola chyb.

Těchto kontrol je celkově 6:

1. `CheckForMultipleComparisonError`: Zakazuje uživateli porovnávání v řadách (např.: `a < b < c`).
2. `CheckForComparisonError`: Zakazuje uživateli použití porovnávacích operátorů v aritmetice (např.: `a = b < c`)
3. `CheckForStatementError`: Kontroluje zda-li uživatel v podmínkách použil porovnávací operace.
4. `CheckForArrayInitializationError`: Kontroluje zda-li je vyplněn rozsah pole, je-li to třeba.
5. `CheckForUninitializedArrayError`: Kontroluje zda-li uživatel pracuje s konkrétním prvkem pole, je-li to třeba.

[7] Z důvodu přehlednosti je většina podmínek ve výpisu nahrazena komentáři, například přiřazovacích tokenů je 7, čili dané podmínky vypadají v tomhle formátu nepřehledně.

6. `CheckForContinueBreakError`: Kontroluje zda-li uživatel nepoužil `continue`, nebo `break` mimo cyklus.

Každá z těchto metod požaduje jako parametr objekt třídy `AST`, nemusí se nutně jednat o listy. Například metoda `CheckForContinueBreakError` pracuje s celým stromem, tento strom je samozřejmě pouhou podmnožinou původního stromu.

3.3.2 Transformace AST

Transformace AST je proces, který připravuje objekt třídy `AST` na další proces, vytvoření CFG. V téhle části můžeme odstranit pomocné tokeny, které byly důležité v třídě `Parser` při tvoření struktury stromu.

Probíhají zde změny struktury tokenů, které nemají reprezentaci v kódu. Ovšem důležitější změny probíhají v přiřazeních, matematických výrazech, voláních funkcí a inicializacích.

- Přiřazení:

Zde se upravují části stromů obsahující tokeny jako jsou `*=`, `/=`, `+=`, `-=`, `++`, či `--`. Metody transformací zde tyhle operace upraví do svých elementárních podob.

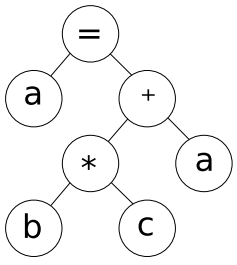
Například operace `a *= b` se upraví na výraz `a = a * b`, všechny obdobné operace se převedou analogicky.

Operace obsahující autoinkrementaci, či autodekrementaci se také transformují do jednoduchých přiřazení. Například `a++` transformujeme na `a = a + 1`. Struktura původního stromu rozlišuje operace jako jsou `a++`, či `++a`. Díky těmto informacím pak vložíme jeho zjednodušenou verzi na správné místo ve stromu.

- Matematické výrazy:

Matematické výrazy jsou problematické, pokud obsahují více jak jeden aritmetický operátor. Takovým výrazem může být například výraz `a = a + b * c`.

Pokud se podíváme na strom tohoto příkazu (Obrázek 1), můžeme vidět že příkaz je deterministicky určen, ale také můžeme vidět mírnou složitost. Pokud chceme přičítat k proměnné `a`, musíme prvně spočítat součin proměnných `b` a `c`.



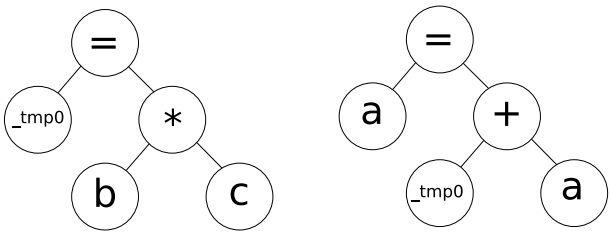
Obrázek 1: AST příkazu: `a = a + b * c`

V téhle chvíli je třeba zavést celočíselnou třídní proměnnou[8] `auxiliaryIndex`, která určuje pomocné proměnné.

Pomocné proměnné mají strukturu `_tmp`⟨X⟩, kde ⟨X⟩ zastupuje hodnotu proměnné `auxiliaryIndex`.

Tato pomocná proměnná nikdy nemůže přijít do konfliktu s proměnnýma, které si zavedl uživatel, protože syntax nepřipouští název proměnných, které začínají jinak než písmeny latinské abecedy (viz. Podkapitola 2.2).

Při zavedení těchto pomocných proměnných se AST z Obrázku 1 transformuje do nového AST', který je znázorněn na Obrázku 2.



Obrázek 2: AST' příkazu: `a = a + b * c`

- Volání funkcí:

V těchto transformacích se jedná o obdobu matematických operací.

Jde o to, že zde kontrolujeme parametry, které vkládáme do volání funkce. V KerLang je totiž přípustné, volat funkci s parametry, jakými jsou funkce, či matematické výrazy (součty více proměnných atp.)

Stejně jako v minulém případě daný strom, aby mohl určit informace, musí prvně vyhodnotit své podstromy.

Volání, které se musí přetransformovat je například:

```
call sum(call sum(5, 8), 5 * 8);
```

Takový příkaz vytvoří strom, který analogicky odpovídá kódu:

```
int _tmp0;
_tmp0 = call sum(5, 8);
int _tmp1;
_tmp1 = 5 * 8;
call sum(_tmp0, _tmp1);
```

[8] Tato třídní proměnná se také využívá při volání funkcí.

- Inicializace:

Poslední případ transformací kódu, upravuje vícenásobné inicializace, jako můžou být například příkazy:

```
int a, b;
int c = 5;
```

Strom vytvořen z tohoto kódu, se ztransformuje do své analogické verze, odpovídající kódu:

```
int a;
int b;
int c;
c = 5;
```

3.3.3 Konkrétní AST

Pro příklad zde uvádím konkrétní AST, který vychází z Výpisu 1, který jsem použil jako příklad v Podkapitole 2.4.

```
Abstract Syntax Tree
TkAST
  TkProcedure
    TkInt
    TkIdentifier: sum
    TkParameters
      TkInt
        TkIdentifier: a
      TkInt
        TkIdentifier: b
    TkBody
      TkReturn
        TkPlus
          TkIdentifier: a
          TkIdentifier: b
  TkProgram
    TkBody
      ...
```

Úroveň odsazení znázorňuje úroveň zanoření listů stromu. Všechny tokeny které jsou odsazeny více než nějaký list `Tk`, jsou potomky tohoto listu `Tk`. Výchozí list stromu se nazývá `TkAST`.

3.4 Graf řídícího toku

CFG je způsob, jak zapsat kód do elementárních kroků, které určují průběh programu, krok po kroku.

Tento graf je tvořen vrcholy a hranami, přičemž každé hraně je přiřazena jedna operace. Touto operací může být například přiřazení do proměnné, volání funkce, aritmetická operace, podmínky, aj.

Vrcholy jsou body grafu, určující pozici v programu, mezi operacemi. Tyto vrcholy využíváme později, během analýz abstraktní interpretace (viz. Kapitola 4). V těchto místech totiž vyhodnocujeme tyto analýzy.

V KerLang je třída reprezentující vytváření CFG nazvána `CFG`.

3.4.1 Třída CFG

Tato třída `CFG` obsahuje všechnu potřebnou logiku, pro vytvoření CFG.

Poté co je vytvořen AST celého programu, centrální část KerLangu inicializuje třídu `CFG`, které předává jako parametr konstruktoru funkci, nebo hlavní funkci (`program`) kódu. Tato funkce je reprezentována objektem třídy `AST`.

Zde můžeme vidět první hlavní rozdíl mezi AST a CFG. AST popisuje celý program, zatím co CFG popisuje jeho části. Každá funkce má svůj CFG, který nemá odkaz na žádný jiný.

Tento objekt třídy `CFG` dále obsahuje všechny informace ohledně funkce, pro kterou je vytvářen. Tyto informace jsou reprezentovány proměnnými:

- `functionName`: Název funkce
- `type`: Návratový typ funkce
- `intParameters`: Všechny celočíselné parametry funkce
- `intArrayParameters`: Všechny parametry, kterými jsou celočíselná pole

Co se týče reprezentace CFG, ten je uchován v proměnné `List<INode> nodes` (viz. Podkapitola 3.4.2).

Dále pro vytvoření potřebujeme proměnnou `actualIndex`, která slouží k udržování informací o tom, s kterou hranou zrovna pracujem. Je totiž třeba každé hraně, kromě hodnoty co je za typ a jaké jsou hodnoty jejich parametrů, přidat ještě informaci o tom, jaký je následující vrchol, na který se má přejít po vyhodnocení tohohle vrcholu. Analogicky si potřebujeme udržovat i informaci o předchozích vrcholech[9].

3.4.2 Interface INode a jeho potomci

Pro vytvoření CFG, potřebujeme mít definované typy uzlů (node), které jsou uloženy v jedné struktuře. Proto využíváme tohoto interface `INode`, který implementují všechny konkrétní uzly.

[9] Tato informace je potřeba v kapitole 4.

Metody a Vlastnosti (Properties) definované tímto interface jsou pouze základní informace o uzlu.

Tyto informace určují typ uzlu, index uzlu, index následujícího uzlu a strukturu indexů ukazující na předchozí uzly. Někteří potomci tyto infromace ještě rozšiřují o další data, která jsou specifická pro typ uzlu.

Celkově rozlišujeme 9 různých typů uzlů dědících z `INode`:

- `Assign`: Uzel popisující přiřazení. Obsahuje navíc proměnné `leftSide`, popisující levou stranu přiřazení (Do které proměnné přiřazujeme), a `rightSide`, popisující pravou stranu, která může být buď proměnnou, či funkcí, a nebo výrazem pouze mezi dvěmi proměnnými (viz. Podkapitola 3.3.3).
- `Break`: Uzel popisující příkaz `break`.
- `Continue`: Uzel popisující příkaz `continue`.
- `Call`: Uzel popisující příkaz `call` (Volání funkce). Obsahuje navíc proměnnou `variables`, která udává parametry volané funkce.
- `If`: Uzel popisující podmínky, spadají zde všechny podmínky ať už `if`, `elseif`, `for`, či `while`. Rozšiřuje původní interface o informace podmínky, proměnná `statement`. Také dále udává informace o větvi, na kterou se přesunout, pokud není podmínka splněná, v proměnné `falseNode`.
- `Initialization`: Uzel popisující inicializaci proměnné. Jako jediný uzel má dva konstruktory. Jeden slouží k inicializaci proměnné, druhý slouží k inicializaci pole. V proměnné `type` si udržujeme informaci o datovém typu. Proměnná `variableName` určuje název proměnné. Poslední přidanou proměnnou je proměnná `arrLength`, určující rozsah pole (pokud inicializujeme pole, v opačném případě zůstává proměnná neinicializovaná).
- `Read`: Uzel popisující čtení z klávesnice, obsahuje navíc informace o proměnných, do kterých něco načítáme. Tyto informace jsou v proměnné `parameters`.
- `Return`: Uzel popisující příkaz `return`. Obsahuje navíc informaci o výrazu, který příkaz vrací. Tato infromace je uložena v proměnné `returnExpression`.
- `Write`: Uzel popisující příkaz pro výpis. Stejně jako uzel `Read`, obsahuje informace o parametrech, které má vypsat. Informace jsou uloženy v proměnné `parameters`.

Všechny další metody obsažené v uzlech jsou pouze pomocné metody sloužící ke správnému vytvoření CFG.

3.4.3 Vytvoření CFG

Celý proces vytvoření CFG z AST začíná metodou `CreateInputNode()`. Tato metoda je volána z konstruktoru třídy `CFG`. V této metodě se nastavují všechny základní informace o funkci, pro kterou se CFG vytváří (viz. Podkapitola 3.4.1.)

Po vytvoření informací charakterizující CFG, je zavolána metoda `CreateNodes()`, tato metoda je centrální částí vytváření CFG. Je volána několikrát během procesu vytváření. Každý blok (ať už blok cyklu, či podmínky) ji volá.

Metoda prvně odstraní nedosažitelné příkazy (nalezení příkazu `return` v hlavním bloku funkce) a poté, směrem od posledního příkazu až po příkaz první, prochází tělem AST.

V tomto procházení se `CFG` dotazuje objektu třídy `AST` o jaký typ tokenu se jedná a podle toho volá příslušnou metodu pro vytvoření uzlu.

Tyto metody se dají rozdělit do dvou skupin. Metody pracující s listy stromu, které žádné další potomky nemají a listy, které mají potomky bloky.

První skupina je pro vytvoření uzlu jednoduší, ze struktury stromu si `CFG` zjistí informace, které potřebuje pro vytvoření. Druhá skupina je složitější z důvodu, že může obsahovat jeden a více bloků, nemůžeme hned říct jaký je index předchozího uzlu, nebo následujícího.

Proto je třeba si v této skupině udržovat informaci o indexu ještě předtím než se zavolá nad novým blokem metoda `CreateNodes()`, kromě této informace, si potřebuje udržovat informace o takzvané `exitNode`. Tato informace je stěžejní při práci s podmínkama, pokud uživatel například napíše kód obsahující složitější strukturu ifů a elseifů, je třeba správné určit následující, respektive předchozí uzel. Bez této informace by jinak poslední příkaz nějakého bloku `elseif` přešel do jiného bloku `elseif`, či `else`, čemuž chceme předejít.

Kromě těchto metod třída `CFG` obsahuje ještě další 4 pomocné metody pro správné vytvoření CFG. Tyto metody jsou:

- `CreateFalseNodeIndex`: Kontroluje zda-li za nějakou podmínkou existuje příkaz `elseif`, nebo `else`. Pokud ano, nastaví proměnnou `falseNode` na tento uzel.
- `CreateTrueNodeIndex`: Obdoba metody `CreateFalseNodeIndex`, je potřeba v situacích, kdy uživatel napíše takzvanou prázdnou podmínku (tělo této podmínky neobsahuje žádný příkaz), kterou je například:

  ```
  if(a > b) { }
  ```

  `CFG` prochází všechny následující `elseif`, či `else` a přiřazuje do `nextNode`, neboli `trueNode` jejich `falseNode`. Jde o to, aby program správně opustil posloupnost podmínek.
- `CheckIfsInLoop`: Tato metoda kontroluje správné opuštění podmínek, či cyklů v cyklu. Tento problém nastává, pokud je podmínka, či cyklus, posledním příkazem v cyklu. Je třeba dodržet aby její `falseNode`, neukazoval mimo cyklus, ale aby se správně vrátil na uzel původního cyklu. Díky tomu může správně začít další iterace cyklu.
- `CheckExitNode`: Podobná situace jako v `CreateTrueNodeIndex`. Nastává když máme posloupnost podmínek v nějaké jiné podmínce, zajišťuje to,

aby se tyto dvě na sobě nezávislé posloupnosti nespojili do jedné v metodách `CreateFalseNodeIndex`, či `CreateTrueNodeIndex`.

3.4.4 Konkrétní CFG

Oproti minulým kapitolám, konkrétní CFG je vytvořen z jiného kódu. Kód který byl použit v Podkapitole 3.3.3 je až příliš přímočarý, proto v Příloze E uvádím jiný, složitější kód[10], ze kterého je tvořen tento konkrétní CFG.

```
Flow control graph:
Function details:
  Type: Void, Name: funkce1
  Int parameters: a, b
    16: Initilization, nextNode: 15, previousNodes:
    15: Initilization, nextNode: 14, previousNodes: 16
    14: Assign, nextNode: 13, previousNodes: 15
    13: Initilization, nextNode: 12, previousNodes: 14
    12: Assign, nextNode: 11, previousNodes: 13
    11: If, trueNode: 10, falseNode: 9, previousNodes: 12
    10: Assign, nextNode: 7, previousNodes: 11
    9: Else, trueNode: 8, falseNode: 7, previousNodes: 11
    8: Assign, nextNode: 7, previousNodes: 9
    7: Assign, nextNode: 6, previousNodes: 8, 9, 10
    6: Initilization, nextNode: 5, previousNodes: 7
    5: Assign, nextNode: 4, previousNodes: 6
    4: Statement, trueNode: 3, falseNode: -1, previousNodes: 0, 5
    3: Assign, nextNode: 2, previousNodes: 4
    2: If, trueNode: 1, falseNode: 0, previousNodes: 3
    1: Assign, nextNode: 0, previousNodes: 2
    0: Assign, nextNode: 4, previousNodes: 1, 2
```

První část CFG uvádí předpis dané funkce. Uvádí její datový typ, název a paramtetry, které přijímá. V druhé číslované části jsou znázorněny konkrétní uzly.

```
index uzlu: Typ uzlu, následující uzel: index následujícího
uzlu, předchozí uzel: indexy předchozích uzlů
```

V uzlech typu `If`, navíc figuruje `trueNode` a `falseNode`, tyto informace znázorňují index následujícího uzlu, pokud je podmínka splněna. V případě `falseNode` znázorňuje index následujícího uzlu, pokud podmínka splněna není.

[10]Tento kód slouží pouze na ukázku, nemá žádný skutečný smysl.

4 Abstraktní interpretace

Abstraktní interpretace je technikou pro statickou analýzu programů. Tyto analýzy se provádějí nad CFG. Samotná abstraktní interpretace je určité zobecnění přístupů pro analýzu datového toku.

Pro samotnou abstraktní interpretaci je potřeba programu, respektive kódu, nad kterým se tyto statické analýzy provádějí. V této práci je jazykem, který definuje kódy, které jsou určeny pro konkrétní analýzu, je KerLang.

Konkrétními zástupci analýz, které spadají do abstraktní interpretace jsou:

- *Live variables*: Tato analýza zkoumá, zda-li hodnota proměnné bude někdy čtena z paměti v průběhu provádění programu. Cílem této analýzy je navrhnout místa programu a proměnné, které například obsahují nějaké hodnoty, ale nikdy nebudou využity, proto není třeba je ani ukládat do paměti.

- *Available expressions*: Tato analýza zkoumá, zda-li program již spočetl nějakou aritmetickou operaci, která se má vyhodnocovat znova, ačkoliv hodnoty jednotlivých dílčích proměnných nebyly změněny. Jinými slovy, navrhuje řešení tak, aby program nepřepočítaval znovu něco, co už má vypočtené v minulých krocích.

- *Very busy expressions*: Tato analýza je podobná analýze *available expressions*, na rozdíl od ní, ale nezkoumá, zda-li nějaká operace byla vypočtena dříve v průběhu programu, ale zkoumá to, jestli bude nějaká operace přepočítávána znova, ještě před tím, než se její hodnota změní.

- *Interval analysis*: Tato analýza ukazuje ve všech bodech programu, rozsah hodnot, kterých může nabývat proměnná. Jinými slovy deterministicky říká v každém bodě, jakou minimální a jakou maximální hodnotu může proměnná mít.

Všechny hodnoty, které abstraktní interpretace uvádí v analýzách jsou pouze aproximacemi, například z *analýzy intervalů* můžeme zjistit, že hodnota nabývá kladných čísel, ve skutečnosti ale může třeba nabývat pouze sudých kladných čísel. Tuto vlastnost *analýzou intervalů* nemůžeme zjistit, ovšem všechny tyto reálné hodnoty spadají do intervalu získaném touto analýzou.

4.1 Svazy

Pro zavedení svazu využívám informací, přebraných z článku: Lattice (order). 2015-29-4. *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation [cit. 2015-05-06]. Dostupné z: https://en.wikipedia.org/wiki/Lattice_(order)

Všechny analýzy spadající do abstraktní interpretace, potřebují svaz, který umožňuje vypočtení výsledků těchto analýz.

Svazem se rozumí množina s relačním uspořádáním:

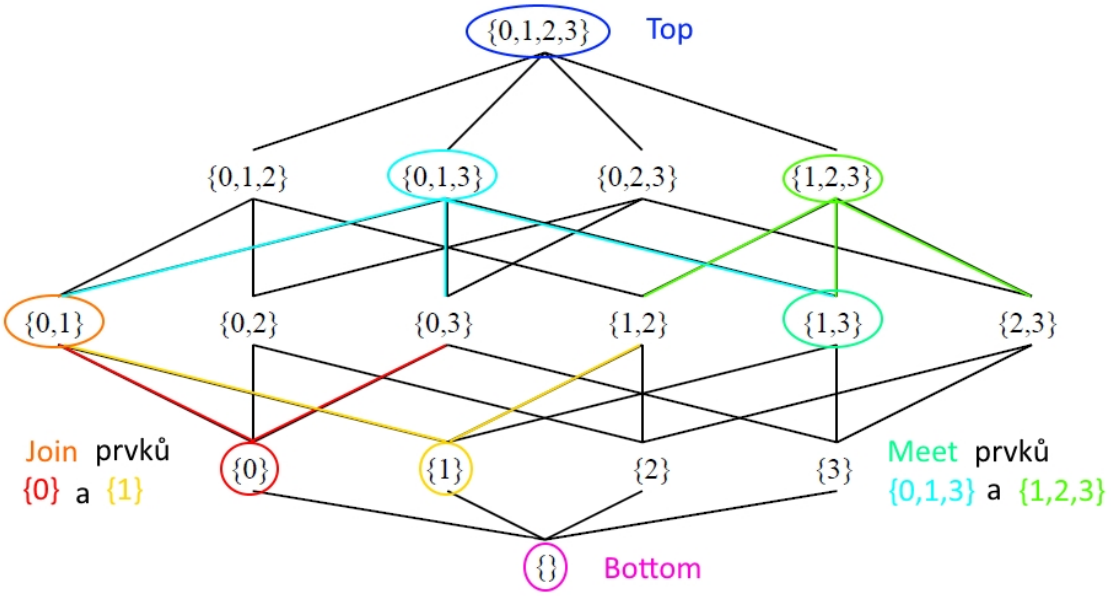
$$L = (S, \sqsubseteq)$$

Tato relace je reflexivní, transitivní a anti-symetrická. Tuto množinu L uspořádanou relací $\sqsubseteq$ nazveme svazem, pokud pro každou dvouprvkovou množinu obsahuje i její supremum a infimum.

Největší prvek svazu se nazývá *Top*, dále jen $\top$, zatím co nejměnší prvek svazu se nazývá *Bottom*, dále jen $\bot$.

Kromě těchto prvků ještě potřebujeme dvě operace kterými je operace *Join*, dále jen $\sqcup$, a operace *Meet*, dále jen $\sqcap$. Analogicky se dá říci, že operace $\sqcup$, má podobný význam jako operace sjednocení při práci s množinami, stejně tak operace $\sqcap$ se dá přirovnat operaci průniku v množinách.

Svazy z matematického hlediska mají ještě další vlastnosti, které se na nich zkoumají, ale pro naše využití v abstraktní interpretaci nám stačí operace relace, $\top$, $\bot$, $\sqcup$ a $\sqcap$. Tyto vlastnosti a operace jsou znázorněny na Obrázku 3.



Obrázek 3: Konkrétní svaz

4.2 Průchod CFG

Typy analýz abstraktní interpretace rozdělujeme do dvou skupin, podle toho jak daný CFG procházejí. Rozlišujeme takzvané dopředné analýzy (forward analysis) a zpětné analýzy (backward analysis). Je potřeba rozlišovat tyto způsoby, protože určité analýzy se dají provádět pouze na určitém typu průchodu.

Oba tyto druhy iterací obsahují 2 instanční proměnné a 2 metody. Tyto proměnné jsou:

1. `CFG graph`: Proměnná obsahující CFG funkce, či programu, který se má analyzovat.

2. `Queue<int> queue`: Fronta indexů pro průchod grafem

Metody těchto tříd dále slouží k připravení objektu pro analýzu (metoda `Prepare()`) a dále ke konkrétnímu volání funkcí svazu, podle typu uzlu, jehož index byl vytažen z fronty (metoda `Analyze()`).

Konkrétní průběh iterací CFG je vykonáván v metodě `Iterate()`, v cyklu `while`.

4.2.1 Dopředná iterace

Dopředné analýzy prochází kódem od vstupního bodu programu, či funkce, až po jeho výchozí bod.

Typický příklad analýzy pro tento způsob průchodu je *analýza intervalů* (*interval analysis*).

Způsob implementace tohoto průchodu spočívá v tom, že si prvně vytvoříme frontu (v implementaci nazvanou `queue`). Tato fronta se naplní indexy od prvního příkazu programu až po jeho poslední.

V prvním kroku se přiřadí konkrétní prvky svazu objektům, které zkoumáme. Do fronty se nic nepřiřazuje, tento krok slouží pouze k zavedení všech uzlů CFG pro budoucí analýzu. Výsledkem tohoto průchodu je objekt připravený na začátek konkrétní analýzy.

V druhém kroku vložíme do fronty vstupní bod programu a všechny jeho navazující uzly.

Z této fronty se postupně získávají indexy ukazující na pozici CFG. Tento uzel CFG je pak poslán jako parametr objektu, který vyhodnocuje námi zvolenou analýzu.

Z tohoto objektu získáváme informace o tom, zda-li se prvek svazu, respektive prvky svazů změnily svoji hodnotu od posledního průchodu tímto uzlem.

Pokud se hodnoty změnily a uzly vycházející z tohoto uzlu nejsou obsaženy ve frontě, tak jej zde přidáme pro další provedení analýzy v daném uzlu. V opačném případě přejdeme k další iteraci cyklu.

Celý tento cyklus probíhá tak dlouho, dokud jsou ve frontě nějaké indexy.

4.2.2 Zpětná iterace

Stejně jako dopředná iterace, zpětná iterace prochází CFG. Ovšem začíná od výchozího bodu funkce, či programu a končí u jeho vstupního bodu.

Typickým příkladem analýzy, využívající tento způsob průchodu CFG je analýza zkoumající *živosti proměnných* (*live variables*).

Co se týče implementace, stejně jako v případě dopředné iterace využíváme fronty (v implementaci nazvanou `queue`). Tato fronta je rozdílná v tom, že si ukládá indexy předchůdců všech uzlů daného uzlu.

Obdobně jako v dopředné iteraci, se tato iterace skládá ze dvou částí. První slouží pouze k nastavení všech uzlů tím, že se jim přiřadí svazy. Poté může začít skutečná analýza.

Tento proces je velice podobný jako v minulé situaci, uzel CFG se předá jako parametr objektu, který vyhodnocuje analýzu. Zpět získáme informaci o změně svazu. Na základě této informace přiřadíme, či nepřiřadíme předchůdce tohoto uzlu do fronty.

Tento proces končí ve chvíli, kdy je fronta prázdná. V této chvíli máme hotovou analýzu, kterou jsme vyhodnocovali.

5 Konkrétní analýzy

Jako konkrétní druhy analýz jsem se rozhodl implementovat *analýzu intervalů* (*interval analysis*) a analýzu *živosti proměnných* (*live variables*).

Tyto analýzy jsem vybral proto, že každá z nich reprezentuje jinou skupinu analýz (viz. Podkapitola 4.2)

Implementaci abstraktní iterpretace jsem pojal jako vytvoření frameworku pro budoucí přidávání dalších typů analýz. Proto každá analýza, kterou chceme vyhodnotit, musí implementovat generický interface `ILattice`. Tento generický interface dědí z jeho negenerické verze. (viz. Výpis 4)

Rozhodl jsem se pro tuto implementaci, protože mi přišla nejjednodušší, co se týče implementace frameworku pro abstraktní interpretaci (jak přidat další analýzu uvádím v Kapitole 6). Tato implementace dovoluje předávat třídu implementující konkrétní analýzu jako parametr třídám `IterationBackward` a `IterationForward`. Díky této vlastnosti není třeba pro každou analýzu implementovat její průchod CFG.

```
public interface ILattice
{
    bool TrueReach { get; }
    bool FalseReach { get; }
    bool First { set; }

    void PrepareInputParameters(List<AST> parameters);
    bool HasChanged(int index);
    List<bool> HasChanged(int index1, int index2);

    bool Operation(object element1, object element2);
    object Join(object element1, object element2);
    object Meet(object element1, object element2);
    void Assign(Assign node, int index, int dest);
    void Break(Break node, int index, int dest);
    void Call(Call node, int index, int dest);
    void Continue(Continue node, int index, int dest);
    void If(If node, int index, int trueDest, int falseDest);
    void Initialization(Initialization node, int index, int dest);
    void Read(Read node, int index, int dest);
    void Return(Return node, int index, int dest);
    void Write(Write node, int index, int dest);
    void Print(CFG graph);
}

public interface ILattice <T>: ILattice where T: IElement
{
    bool Operation(T element1, T element2);
    T Join(T element1, T element2);
    T Meet(T element1, T element2);
}
```

Výpis 4: Interface `ILattice`

Vlastnosti `TrueReach`, `FalseReach` a `First` slouží k informacím, které slouží průchodu CFG. `TrueReach` a `FalseReach` slouží analýzám, které vyhodnocují jestli je daná větev podmínky dosažitelná. Pokud je tato informace pro nějaké jiné analýzy nepodstatná defaultně má hodnotu `true`. Vlastnost `First` slouží k první iteraci uzlu.

Metody `Join()`, `Meet()` a `Operation()` slouží k implementaci operací nad svazem, se kterým se pracuje v konkrétní analýze.

Metody `Assign()`, `Break()`, `Call()`, `Continue()`, `If()`, `Initialization()`, `Read()`, `Return()` a `Write()` slouží k implementaci konkrétní analýzy nad daným uzlem (názvy metod, korespendují s typem uzlu).

Metoda `PrepareInputParameters()` spracovává parametry funkce. Metody `HasChanged()` vrací informaci o tom, zda-li se uzlu na daném indexu přiřadily jiné prvky svazu. Jeho přetížená verze se využívá v uzlech, které se větví (v KerLang to jsou pouze objetky třídy `If`).

Poslední metoda `Print()` slouží k výpisu analýzy, je volána úplně jako poslední. Parametr `Graph` zde slouží pouze k přehlednosti, vypisuje se zde informace o uzlu a výsledek analýzy v tomto uzlu.

Interface `IElement` nedefinuje žádné metody, ani vlastnosti. Elementy totiž závisí na konkrétní analýze a podle ní jsou definovány. Důvod proč tyto třídy musí implementovat tento interface je pouze z hlediska bezpečnosti přístupu k analýze (encapsulation).

5.1 Analýza intervalů

Analýza intervalů vyhodnocuje horní a dolní mez hodnot pro každou proměnnou v každém kroku programu.

Tato analýza je vyhodnocována postupně směrem od začátku funkce, či programu, až po jeho konec. Třída, která tuto analýzu vyhodnocuje je v implementaci nazvána `IntervalLattice`. Generikum, které je předáváno této třídě je nazváno `IntervalElement`.

`IntervalElement` reprezentuje interval hodnot, ve formě `(min, max)`. Tyto hodnoty (`min` a `max`), ale nemusí být čísla, pracujeme také s hodnotama jakými jsou $+\infty$ a $-\infty$. Z tohoto důvodu má tato třída 4 různé konstruktory.

1. `IntervalElement(BigInteger min, BigInteger max)`: Tento konstruktor nastavuje interval, který má svou horní a dolní mez číslo.

2. `IntervalElement(ElementIntervalInfo min, BigInteger max)`: Tento konstruktor nastavuje interval, který má svou dolní mez $-\infty$ a horní mez číslo.

3. `IntervalElement(BigInteger min, ElementIntervalInfo max)`: Tento konstruktor nastavuje interval, který má svojí dolní mez číslo a horní mez rovnu $+\infty$.

4. `public IntervalElement(ElementIntervalInfo info)`: Tento konstruktor může definovat dva různé intervaly, první z nich nastává v situaci, kdy výčet

`info`, má hodnotu `bottom`, udává že daný interval, respektive proměnná na kterou se tento interval váže, nemá žádnou hodnotu, není definovaný. V opačném případě může parametr `info` nabývat hodnoty `top`, v tomto případě vzniká interval který má svou dolní mez rovnu $-\infty$ a horní mez rovnu $+\infty$.

Veškerá analýza je udržovaná v proměnné `List<AbstractContextInterval>[] contexts`. Přičemž `AbstractContextInterval` je strukturou obsahující `IntervalElement element`, reprezentující daný interval a `String name` reprezentující jméno proměnné, ke které se interval váže.

Operace $\sqcup$ vytváří interval, jehož minimum je rovno nejmenšímu minimu těchto dvou intervalů, nad kterými je tato operace použita. Stejně tak maximum je rovno největšímu maximu těchto intervalů.

$$(\texttt{min1}, \texttt{max1}) \sqcup (\texttt{min2}, \texttt{max2}) =$$

- $(\texttt{min1}, \texttt{max1})$: $\texttt{min1} \leq \texttt{min2} \wedge \texttt{max1} \geq \texttt{max2}$
- $(\texttt{min1}, \texttt{max2})$: $\texttt{min1} \leq \texttt{min2} \wedge \texttt{max1} \leq \texttt{max2}$
- $(\texttt{min2}, \texttt{max1})$: $\texttt{min1} \geq \texttt{min2} \wedge \texttt{max1} \geq \texttt{max2}$
- $(\texttt{min2}, \texttt{max2})$: $\texttt{min1} \geq \texttt{min2} \wedge \texttt{max1} \leq \texttt{max2}$

Operace $\sqcap$ vytváří interval, jehož minimum je větší minimum z daných intervalů, ale zároveň musí být menší než maximum druhého intervalu. Jinými slovy tato hodnota musí být obsažena v obou intervalech. Stejně tak pro maximum platí, že je to menší maximum daných intervalů, které je ovšem větší než minimum druhého intervalu.

$$(\texttt{min1}, \texttt{max1}) \sqcap (\texttt{min2}, \texttt{max2}) =$$

- $\bot$: $\texttt{max1} \leq \texttt{min2} \vee \texttt{max2} \leq \texttt{min1}$
- $(\texttt{min1}, \texttt{max2})$: $\texttt{min1} \geq \texttt{min2} \wedge \texttt{min1} \leq \texttt{max2}, \texttt{max2} \leq \texttt{max1} \wedge \texttt{max2} \geq \texttt{min1}$
- $(\texttt{min2}, \texttt{max1})$: $\texttt{min2} \geq \texttt{min1} \wedge \texttt{min2} \leq \texttt{max1}, \texttt{max1} \leq \texttt{max2} \wedge \texttt{max1} \geq \texttt{min2}$
- $(\texttt{min1}, \texttt{max1})$: $\texttt{min1} \geq \texttt{min2} \wedge \texttt{max1} \leq \texttt{max2}$
- $(\texttt{min2}, \texttt{max2})$: $\texttt{min2} \geq \texttt{min1} \wedge \texttt{max2} \leq \texttt{max1}$

Kromě těchto operací, které jsem zmínil již v Podkapitole 4.1, je třeba ještě nadefinovat speciální operaci *Widening*. Během vyhodnocování analýzy je program totiž procházen podobně, jako by byl procházen při skutečném spuštění. Jinými slovy pokud chceme analyzovat určitou větev podmínky, musíme prvně získat informaci o tom, že podmínka je dosažitelná (tento celý proces popisuji detailněji v další části této kapitoly). Díky tomuhle přístupu nám může vzniknout situace v cyklech (například v nekonečných cyklech bez podmínky), kde se v nějakém kroku cyklu mění hodnoty proměnných, takže analýza nemůže skončit a vyhodnocovala by se nekonečně dlouho. Proto zavádíme operaci *Widening*, která po určitém počtu iterací cyklu (tento počet závisí na nastavení "settings.txt", viz. Kapitola 3) aproximuje minimální, respektive maximální hodnotu intervalu k $-\infty$, respektive k $+\infty$.

$$(\texttt{min1}, \texttt{max1})\ \textit{Widening}\ (\texttt{min2}, \texttt{max2}) =$$

- $(\texttt{min1}, \texttt{max1})$: $\texttt{min1} = \texttt{min2} \wedge \texttt{max1} = \texttt{max2}$
- $(\texttt{min1}, +\infty)$: $\texttt{min1} = \texttt{min2} \wedge \texttt{max1} < \texttt{max2}$
- $(-\infty, \texttt{max1})$: $\texttt{min1} > \texttt{min2} \wedge \texttt{max1} = \texttt{max2}$
- $(-\infty, +\infty)$: $\texttt{min1} > \texttt{min2} \wedge \texttt{max1} < \texttt{max2}$

Poté co máme nadefinované všechny tyto operace, je třeba ještě nadefinovat aritmetiku pro práci s intervaly, k tomu slouží metody

- `IntervalElement Plus(IntervalElement element1, IntervalElement element2)`
- `IntervalElement Minus(IntervalElement element1, IntervalElement element2)`
- `IntervalElement Divide(IntervalElement element1, IntervalElement element2)`
- `IntervalElement Multiple(IntervalElement element1, IntervalElement element2)`

Ve všech těchto metodách je potřeba zajistit, aby byly v intervalech obsaženy všechny možnosti, kterých může daná proměnná nabývat. Proto se matematická operace provede nad všemi kombinacemi `min` a `max`. Například u operace násobení, nemůžeme říct, že součin `min1` a `min2` je nejmenším prvkem nového intervalu. Pokud totiž obě tyto hodnoty jsou záporné, získáme kladný výsledek, který je větší než jakékoliv z těchto minim.

Poslední pomocnou metodou pro vyhodnocení analýzy, je metoda, která pracuje s podmínkami. Tato metoda má předpis `void IfOperation(ref AbstractContextInterval newContextTrue, ref AbstractContextInterval newContextFalse, AbstractContextInterval rightSide, String operation)`. Z důvodu komplexnosti této metody, uvádím její princip v tabulce, v Příloze D. Tato tabulka popisuje obecné skupiny, které můžeme porovnávat a výsledky, které udávají interval, který je větší, menší, roven, atp. než jiný interval.

V této metodě je také třeba spočítat negaci dané operace, tento výsledek je pak použit pro větev, kterou program prochází, pokud podmínka splněna není.

Kromě těchto pomocných metod, třída `IntervalLatice` ještě obsahuje pomocné proměnné. Proměnné `contextBefore1` a `contextBefore2`[11], v sobě nesou informace o analýze v daném kroku programu, ještě před tím, než jsme začali tyto hodnoty přepočítávat. Poté se na základě změn určí, zda-li je třeba dalších iterací CFG. Toto porovnání zprostředkovává metoda `bool HasChanged()`.

[11] Z důvodu větvení v grafu (objekty třídy `If`), je třeba si v těchto situacích uchovávat původní informace o obou uzlech, na které uzel `If` ukazuje.

Poté co máme naimplementovány všechny tyto metody, je třeba ještě doimplementovat interface `ILattice`. Metody které chybí doimplementovat jsou metody pro práci s konkrétními uzly CFG. Tyto uzly můžeme rozdělit do dvou skupin:

- Uzly které nijak nemění hodnoty intervalů: Do této skupiny spadají uzly tříd: `Break`, `Continue`, `Initialization`, `Write`, `Call` a `Return`. Tyto skupiny pouze kopírujou hodnoty navázané na jejich uzel, na uzel následující.

```
public void Call(Call node, int index, int dest)
{
    contextBefore1 = contexts[dest];

    JoinOperation(index, dest);

    return ;
}
```

Výpis 5: Metoda `Call` neměnící hodnotu intervalů

- Uzly které mění hodnoty intervalů: Do této skupiny spadají uzly tříd: `Assign`, `Read` a `If`. Tyto metody mají dvě části. První část slouží k nastavení svazů při prvním průchodu (Ještě předtím, než se začne vyhodnocovat skutečná analýza), v této části se všechny svazy nastaví na $\bot$. V druhé části se vyhodnocuje již skutečná analýza.

  Metoda `Assign` vytváří intervaly zapomocí metod `Plus`, `Minus`, `Multiple` a `Divide` v případě, že operace přiřazení používá aritmetiku. V opačném případě (přiřazujeme-li proměnnou) pouze vytvoří kopii intervalu, který patří proměnné, která má být přiřazena. Pokud přiřazujeme konstantu, vytvoří nový interval, který obsahuje pouze hodnotu této konstanty.

  Metoda `Read` vytváří intervaly nabývající pouze hodnoty $\top$.

  Metoda `If` vytváří intervaly zapomocí metody `IfOperation`. Tato metoda vytváří hned několik intervalů, pokud například porovnáváme `a < b`. Musíme také vyhodnotit analýzu pro proměnnou `b` ve formě `b > a`. Stejně tak musíme vyhodnotit negaci těchto podmínek pro vyhodnocení rozsahu intervalu, když podmínka splněna není. Metoda dále zkoumá, zda-li hodnota intervalu nabývá hodnot jiných než $\bot$. V případě, že interval nabývá této hodnoty, nepřenáší se do dané větve žádné informace, tato větev je totiž nedosažitelná. Všechny prvky svazu zde zůstanou nastaveny na $\bot$, z prvního průchodu CFG.

Další informace na toto téma se nacházejí v úryvku Knihy [1], nebo v Poznámkách [2].

5.1.1 Konkrétní analýza intervalů

Pro příklad zde uvádím konkrétní výsledek *analýzy intervalů*. Kód, který je použit pro tuto analýzu je znázorněn v Příloze E.

```
 Analysis Interval
  a: (Top) b: (Top)
16: Initilization, nextNode: 15, previousNodes:
  a: (Top) b: (Top)
15: Initilization, nextNode: 14, previousNodes: 16
  a: (Top) b: (Top)
14: Assign, nextNode: 13, previousNodes: 15
  a: (Top) b: (Top) c: ( 20 ; 20)
13: Initilization, nextNode: 12, previousNodes: 14
  a: (Top) b: (Top) c: ( 20 ; 20)
12: Assign, nextNode: 11, previousNodes: 13
  a: (Top) b: (Top) c: ( 20 ; 20) h: (Top)
11: If, trueNode: 10, falseNode: 9, previousNodes: 12
  a: (Top) b: (Top) c: ( 20 ; 20) h: (-inf ; 19)
10: Assign, nextNode: 7, previousNodes: 11
  a: (Top) b: (Top) c: ( 20 ; 20) h: (20 ; +inf)
9: Else, trueNode: 8, falseNode: 7, previousNodes: 11
  a: (Top) b: (Top) c: ( 20 ; 20) h: (20 ; +inf)
8: Assign, nextNode: 7, previousNodes: 9
  a: (Top) b: ( 20 ; 20) c: ( 20 ; 20) h: (Top)
7: Assign, nextNode: 6, previousNodes: 8, 9, 10
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top)
6: Initilization, nextNode: 5, previousNodes: 7
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top)
5: Assign, nextNode: 4, previousNodes: 6
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top) i: ( -10 ; -5) u: ( 15 ; 15)
4: Statement, trueNode: 3, falseNode: -1, previousNodes: 0, 5
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top) i: ( -9 ; -5) u: ( 15 ; 15)
3: Assign, nextNode: 2, previousNodes: 4
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top) i: ( -9 ; -5) u: ( 15 ; 15)
2: If, trueNode: 1, falseNode: 0, previousNodes: 3
  a: (Bottom) b: (Bottom) c: (Bottom) h: (Bottom) i: (Bottom) u: (Bottom)
1: Assign, nextNode: 0, previousNodes: 2
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top) i: ( -9 ; -5) u: ( 15 ; 15)
0: Assign, nextNode: 4, previousNodes: 1, 2
  a: ( 5 ; 5) b: (Top) c: ( 20 ; 20) h: (Top) i: ( -10 ; -10) u: ( 15 ; 15)
```

Výsledky analýzy ve formátu `proměnná: (min,max)` se vážou k uzlu, který se nachází na následujícím řádku. Pokud na nějaký uzel ukazuje více jiných uzlů, je v tomto uzlu analýza sloučena ze všech dílčích analýz, které byly spočteny těmito předchůdci.

5.2 Živost proměnných

Analýza *živosti proměnných*, zkoumá zda-li je proměnná živá, či nežívá. Živá proměnná je taková, jejíž hodnota bude vyžadována v následujících krocích programu.

Tato analýza je zástupcem analýz, využívajících zpětné iterace. V implementaci je tato analýza vyhodnocována třídou `LivenessLatice`. Generikum, které je využito pro inicializaci této třídy, se nazývá `LivenessElement`.

Oproti *analýze intervalů*, je svaz mnohem jednodušší. Skládá se pouze ze dvou hodnot. $\bot$ v situaci, kdy je proměnná neživá, $\top$ v situaci, kdy je proměnná živá.

Operace $\sqcup$ má také velice jednoduchou implementaci. Tato operace vytváří element, který je živý, pokud jeden z elementů nad kterými je tato operace zavolána, nabývá hodnoty $\top$.

`element1` $\sqcup$ `element2` =

- $\top$ (proměnná je živá): element1, nebo element2 nabývá hodnoty $\top$.
- $\bot$ (proměnná je neživá): element1 a element2 nabývají hodnoty $\bot$.

Tato metoda `Join`, je jediná pomocná metoda, která je potřeba pro vyhodnocení této analýzy. Není zde potřeba operace $\sqcap$, ve všech uzlech CFG je třeba hodnoty pouze skládat za pomocí operace $\sqcup$. Stejně tak operace *Widening* není v této analýze nijak potřeba, analýza totiž nezkoumá průchody a dosažitelnosti grafu. Díky tomuhle přístupu nenastáva situace pro možnost zacyklení iterace.

Stejně jako u *analýzy intervalů*, využíváme pomocné proměnné `private List<AbstractContextLiveness> contextBefore`, která obsahuje svazy přiřazené proměnným v právě analyzovaném uzlu CFG. Za pomocí metody `bool HasChanged()` informuje třídu, zprostředkovávající iteraci CFG, o změnách v analyzovaném uzlu. V případě že žádné změny neproběhly a tento uzel již předal informace o všech svých předchůdcích v nějaké minulé iteraci, nepřidává do fronty indexů své předchůdce. V opačném případě své předky do fronty přidá.

Co se týče implementace metod, pro práci s konkrétními uzly, můžeme tyto metody rozdělit do dvou skupin. První skupina obsahuje metody, které nerozhodujou o živosti proměnných, zatím co druhá skupina obsahuje metody, které o živosti proměnných rozhodují.

- Metody, které nerozhodují o živosti proměnných: `Initialization`, `Continue` a `Break`. Implementace těchto metod, je totožná jako u *analýzy intervalů*, viz. Výpis 5.
- Metody, které rozhodují o živosti proměnných: Všechny tyto metody obsahují dvě části. První z těchto části je pro všechny tyto metody společná[12]. Tato část se vykonává pouze v prním průchodu daného uzlu, a slouží k nastavení svazu pro analýzu. Během tohoto nastevení, se všechny proměnné nastavují na hodnotu $\bot$.

[12]Tyto metody se liší zápisem kódu, kvůli struktuře daného uzlu. Jejich cíl je ale stejný.

Tuto skupinu metod můžeme rozdělit na další dvě podskupiny. Metody, které nastavují proměnné jako neživé a metody, které nastavují proměnné jako živé.

Metody `Call()` (proměnné, které jsou předávaný jako paramtry funkci), `Write()` (proměnné, které se mají vypisovat), `Return()` (proměnné, které jsou použity v příkazu return), `If()` (všechny proměnné, figurující v podmínkách) a `Assign()` (proměnné, které figurují na pravé straně přiřazení.) Všechny tyto uvedené proměnné se nastaví jako živé. Postupně se tato informace šíří do předchůdců daného uzlu.

Jediná možnost jak změnit vlastnost proměnných z *živá* na *neživá*, je možné pouze za pomocí metod `Assign()` (levá strana přiřazení) a `Read()` (Všechny proměnné, které jsou v parametrech tohoto příkazu). Důvod proč tyto metody nastavují vlastnost proměnné na *neživá*, je takový, že v těchto operacích je hodnota, kterou tato proměnná má, přepsána hodnotou jinou.

Další informace na toto téma se nacházejí v Poznámkách [2].

5.2.1 Konkrétní analýza živosti proměnných

Pro příklad zde uvádím výsledek analýzy *živosti proměnných*. Kód, který byl použit pro tuto analýzu je znázorněn v Příloze E).

```
 Analysis Liveness
  i: Alive a: Alive u: Dead b: Dead c: Dead h: Dead
16: Initilization, nextNode: 15, previousNodes:
  i: Alive a: Alive u: Dead b: Dead c: Dead h: Dead
15: Initilization, nextNode: 14, previousNodes: 16
  i: Alive a: Alive u: Dead b: Dead c: Dead h: Dead
14: Assign, nextNode: 13, previousNodes: 15
  i: Alive a: Alive u: Dead b: Dead c: Alive h: Dead
13: Initilization, nextNode: 12, previousNodes: 14
  i: Alive a: Alive u: Dead b: Dead c: Alive h: Dead
12: Assign, nextNode: 11, previousNodes: 13
  i: Alive a: Dead u: Dead b: Dead c: Alive h: Alive
11: If, trueNode: 10, falseNode: 9, previousNodes: 12
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
10: Assign, nextNode: 7, previousNodes: 11
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
9: Else, trueNode: 8, falseNode: 7, previousNodes: 11
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
8: Assign, nextNode: 7, previousNodes: 9
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
7: Assign, nextNode: 6, previousNodes: 8, 9, 10
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
6: Initilization, nextNode: 5, previousNodes: 7
  i: Dead a: Dead u: Dead b: Dead c: Dead h: Dead
5: Assign, nextNode: 4, previousNodes: 6
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
4: Statement, trueNode: 3, falseNode: -1, previousNodes: 0, 5
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
3: Assign, nextNode: 2, previousNodes: 4
  i: Alive a: Dead u: Alive b: Dead c: Dead h: Dead
2: If, trueNode: 1, falseNode: 0, previousNodes: 3
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
1: Assign, nextNode: 0, previousNodes: 2
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
0: Assign, nextNode: 4, previousNodes: 1, 2
  i: Alive a: Dead u: Dead b: Dead c: Dead h: Dead
```

Výsledky analýzy jsou ve formátu `proměnná: Alive/Dead`. `Alive` značí že proměnná je v daném uzlu živá, `Dead` značí, že proměnná je v daném uzlu neživá. Výsledky analýzy se vážou k uzlu, který se nachází na následujícím řádku.

6 Práce s programem

6.1 Spuštění programu

V této práci se syntaktická analýza KerLang (včetně vytvoření CFG) a analýzy abstraktní interpretace spouští neodděleně. (Není možné vytvořit jen CFG, bez toho aniž by byl zanalyzován konkretními analýzami abstraktní interpretace, viz. Kapitola 5).

Pro spuštění programu je třeba napsat kód v jazyce KerLang, který je uložen v textovém souboru (práce obsahuje také několik konkrétních příkladů jendoduchých kódů, na kterých byla práce vyzkoušena).

Dále je třeba nastavit textový soubour settings, ve kterém se nastaví cesty na kód, který má být předán programu pro další práci. Dále také cesta na adresář, do kterého se analýzy (včetně CFG a AST) uloží. Pokud soubor neexistuje, program jej vytvoří. V případě, že se soubor nepodařilo vytvořit, je výstup vypsán do konzole a uživatel je také upozorněn, že se data nepodařilo uložit. Soubor settings se nachází na adrese "KermanLang\Settings\settings.txt".

Pokud je napsán kód v tomto jazyce a soubor settings je také nastaven, můžeme spustit program. Cesta k souboru, který vytvoří z kódu CFG a zanalyzuje jej, je následující — "KermanLang\bin\Release\KerLang.exe".

Po spuštění programu, je uživatel v konzoli informován o práci programu. Ve složce, kterou uživatel napsal do souboru settings, se vytvoří 4 nové textové soubory. Tyto soubory obsahují AST, CFG, analýzu intervalů a analýzu živosti proměnných.

6.2 Přidání analýzy do frameworku

Pro přidání nové analýzy spadající do abstraktní interpretace, je třeba naimplementovat třídu, která implementuje rozhraní `IElement`. V této třídě je reprezentován konkrétní prvek svazu.

Dále je třeba implementovat třídu, která implementuje rozhraní `ILattice<T> : ILattice where T : IElement`. V této třídě se implementuje logika celé analýzy, viz. 5.

Poté co je naimplementována tato třída, je třeba ještě přidat jeden příkaz, který informuje program, že má začít kód analyzovat i touto novou analýzou. Prvně je třeba rozhodnout jaký druh iterace daná analýza využívá.

Poté je třeba v cyklu, nachází se v třídě `Program`, který prochází všechny CFG vytvořit objekt reprezentující danou iteraci (buď to je objekt třídy `IterationForward`, nebo `IterationBackward`). Tomuto objektu předá jako parametr konstruktoru tento nový objekt třídy pro analýzu.

Daná iterace pak prochází CFG a nad všemi jeho uzly provádí analýzu implementovanou v objektu, který jsme předali v konstruktoru.

7 Závěr

V této bakalářské práci jsem naimplementoval framewrok pro abstraktní interpretaci.

Abych mohl tento framework implementoval, musel jsem prvně implementovat programovací jazyk KerLang. V tomto jazyce jsem napsal programy, které se analyzují touto abstraktní interpretací.

Abych mohl naprogramovat tento jazyk, musel jsem nastudovat problematiku parsování, AST a CFG. Poté co jsem tento jazyk naimplementoval, začal jsem implementovat framework pro abstraktní interpretaci.

Jako konkrétní příklady abstraktní interpretace jsem použil *analýzu intervalů* a *analýzu živosti proměnných*. Zdroje z kterých jsem čerpal, při implementaci těchto analýz jsou [1] a [2].

Program je navržen tak, aby do něj bylo možné přidat jednoduše další analýzy. Tento proces vyžaduje implementaci třídy, specifikující analýzu, tato třída dědí z generického interface.

Jakub Kermaschek

8 Reference

[1] Patrik Cousot, Rathia Cousot, *Static determination of dynamic properties of programs*, 2nd International Symposium on Programming, Paris, pp. 106-130, 1976

[2] Michael I. Schwartzbach, *Lecture Notes on Static Analysis*, Basic Research in Computer Science, University of Aarhus, Denmark, 2008

A Použitý software

Tato sekce obsahuje výčet softwaru, který byl při tvorbě této práce použit.

- Microsoft Visual Studio 2013 Ultimate with Update 4 (x86)
- Microsoft Windows Professional 8.1 (x86)
- TeXstudio 2.9.4

B Druhy tokenů jazyka KerLang

0.	Token	Název tokenu	Význam
1.	`+`	TkPlus	Plus
2.	`++`	TkPlusPlus	Inkrementace o 1
3.	`+=`	TkPlusEqual	Přičtení k levé straně
4.	`-`	TkMinus	Mínus
5.	`-`	TkMinusMinus	Dekrementace o 1
6.	`-=`	TkMinusEqual	Odečtení od levé strany
7.	`*`	TkMultiple	Krát
8.	`*=`	TkMultipleEqual	Znásobení levé strany
9.	`/`	TkDivide	Děleno (celočíselné)
10.	`/=`	TkDivideEqual	Podělení z levé strany
11.	`,`	TkComma	Čárka
12.	`;`	TkSemicolon	Středník
14.	`(`	TkLeftParenthesis	Levá závorka
15.	`)`	TkRightParenthesis	Pravá závorka
16.	`{`	TkLeftBrace	Levá složená závorka
17.	`}`	TkRightBrace	Pravá složená závorka
18.	`[`	TkLeftBracket	Levá hranatá závorka
19.	`]`	TkRightBracke	Pravá hranatá závorka
20.	`for`	TkFor	Začátek cyklu for
21.	`if`	TkIf	Začátek podmínky
22.	`elseif`	TkElseIf	Další podmínky, pokud žádná předchozí nebyla splněna
23.	`else`	TkElse	Co se má stát, když žádná podmínka není splněna
24.	`while`	TkWhile	Začátek cyklu while, konec cyklu do-while
25.	`break`	TkBreak	Ukončení celého cyklu
26.	`continue`	TkContinue	Začátek nové iterace cyklu
27.	`do`	TkDo	Začátek cyklu Do-While
28.	`==`	TkEqual	Rovná se
29.	`!=`	TkNotEqual	Nerovná se
30.	`>`	TkGreater	Je větší než
31.	`<`	TkLess	Je menší než
32.	`>=`	TkGreaterEqual	Je větší, nebo rovno než
33.	`<=`	TkLessEqual	Je menší, nebo rovno než
34.	`&`	TkAnd	Booleanovský and
35.	`|`	TkOr	Booleanovský or
36.	`=`	TkAssign	Přiřazení

37.	`!`	TkNegation	Negace
38.	`"`	TkQuotes	Uvozovky, uvádějí začátek a konec řetězce
39.	`write`	TkWrite	Výpis na obrazovku
40.	`read`	TkRead	Čtení z klávesnice
41.	`Function`	TkFunction	Procedura, či funkce
42.	`program`	TkProgram	Hlavní funkce programu
43.	`int`	TkInt	Celočíselný datový typ
44.	`void`	TkVoid	Datový typ, který informuje o tom, že funkce nic nevrací
45.	`//`	TkComment	Začátek komentáře
46.	`return`	TkReturn	Ukončení funkce, či procedury
47.	`call`	TkCall	Volání funkce, či procedure
48.		TkString	Řetězec, který se vyskytuje jako parametr příkazu `write`. Může obsahovat jakýkoliv znak
49.		TkIdentifier	Proměnná, musí začínat písmenem Abecedy (A-Z, nebo a-z), dále může následovat libovolná kombinace A-Z, a-z, nebo 0-9
50.		TkConstant	Konsanta, musí začínat číslicí 1-9, dále může následovat libovolná kombinace 0-9
51.		TkEOF	Konec souboru s kódem
52.		TkError	Chyba v kódu
53.		TkAST	Začátek AST
54.		TkParameters	Parametry funkcí a procedur
55.		TkArr	Proměnná je pole
56.		TkParameter	Konkrétní parametr funkce, či procedury
57.		TkBody	Tělo funkce, procedury, podmínek, či cyklů
58.		TkIndex	Index označující deklaraci pole (ne konkrétní index pole)
59.		TkStatement	Booleovský výraz
60.		TkForInit	Inicializace proměnných v cyklu for
61.		TkForArithmetics	Aritmetika v cyklu for
62.		TkArithmetic	Aritmetika
63.		TkExpression	Aritmetický výraz

C Gramatika KerLang

BG je určena množinou terminálů, množinou neterminálů, množinou přepisovacích pravidel a počátečním neterminálem.

V KerLang je počáteční neterminál BG nazván LANG.

- Množina neterminálů = {LANG, TYPE, INIT, INITMORE, ARR, OPERATIONS, DECLARATIONVARIABLE, DECLARATION, DECLARATIONMORE, DECLARATIONARR, DECLARATIONARRMORE, VALUEWRITE, VALUEWRITEMORE, VARIABLEREAD, VARIABLEREADMORE, VARIABLECALL, VARIABLECALLMORE, ELSEIF, ELSE, FORINIT, FOREXPRESSION, FORARITHMETICS, RETURN, ARITHMETIC, ARITHMETICMORE, ASSIGNS, EQUALITY, ADDITIVES, MULTIPLICATIONS, INCDEC, EXPRESSION, ADD, MULTIPLE, IDS}
- Množina terminálů = {`function`, ⟨*identifier*⟩, `(`, `)`, `{`, `}`, `program`, `void`, `int`, `,`, `[`, `]`, `;`, `write`, `read`, `call`, `if`, `for`, `while`, `do`, `return`, `break`, `continue`, `=`, `"`, `elseif`, `else`, ⟨*string*⟩, `+=`, `-=`, `*=`, `/=`, `>`, `>=`, `<`, `<=`, `==`, `!=`, `+`, `-`, `*`, `/`, `++`, `--`, ⟨*constant*⟩, ⟨*eof*⟩, ε[13]}
- Množina přepisovacích pravidel = {

LANG	→	`function` TYPE ⟨*identifier*⟩ `(` INIT `)` `{` OPERATIONS `}` LANG
	|	`program` `{` OPERATIONS `}` LANG
	|	⟨*eof*⟩
TYPE	→	`void`
	|	`int`
INIT	→	`int` ⟨*identifier*⟩ ARR INITMORE
	|	ε
INITMORE	→	`,` `int` ⟨*identifier*⟩ ARR INITMORE
	|	ε
ARR	→	`[` EXPRESSION `]`
	|	ε
OPERATIONS	→	`int` DECLARATIONVARIABLE `;` OPERATIONS
	|	`write` `(` VALUEWRITE `)` `;` OPERATIONS
	|	`read` `(` VARIABLEREAD `)` `;` OPERATIONS

[13] Speciální znak ε zastupuje takzvané "prázdné slovo."

| `call` ⟨*identifier*⟩ `(` VARIABLECALL `)` `;` OPERATIONS
| `if (` EXPRESSION `) {` OPERATIONS `}` ELSEIF ELSE OPERATIONS
| `for (` FORINIT `;` FOREXPRESSION `;` FORARITHMETIC `) {` OPERATIONS `}` OPERATIONS
| `while (` EXPRESSION `) {` OPERATIONS `}` OPERATIONS
| `do {` OPERATIONS `} while (` EXPRESSION `)` OPERATIONS
| `return` RETURN `;` OPERATIONS
| `break ;` OPERATIONS
| `continue ;` OPERATIONS
| ARITHMETIC `;` OPERATIONS
| ε

DECLARATIONVARIABLE → ⟨*identifier*⟩ DECLARATION

DECLARATION → `=` EXPRESSION DECLARATIONMORE
| DECLARATIONMORE
| ARR DECLARATIONARR

DECLARATIONMORE → `,` DECLARATIONVARIABLE
| ε

DECLARATIONARR → `= {` EXPRESSION DECLARATIONARRMORE `}`
| DECLARATIONARRMORE
| ARR DECLARATIONARR

DECLARATIONARRMORE → `,` EXPRESSION DECLARATIONARRMORE
| ε

VALUEWRITE → `"` ⟨*string*⟩ `"` VALUEWRITEMORE
| EXPRESSION VALUEWRITEMORE
| ε

VALUEWRITEMORE → `,` VALUEWRITE
| ε

VARIABLEREAD → ⟨*identifier*⟩ ARR VARIABLEREADMORE
| ε

VARIABLEREADMORE → , VARIABLEREAD
| ε

VARIABLECALL → EXPRESSION VARIABLECALLMORE
| ε

VARIABLECALLMORE → , VARIABLECALL
| ε

ELSEIF → `elseif` (EXPRESSION) { OPERATIONS } ELSEIF
| ε

ELSE → `else` { OPERATIONS }
| ε

FORINIT → `int` DECLARATIONVARIABLE
| DECLARATIONVARIABLE
| ε

FOREXPRESSION → EXPRESSION
| ε

FORARITHMETICS → ARITHMETIC ARITHMETICMORE
| ε

RETURN → EXPRESSION
| ε

ARITHMETIC → ⟨*identifier*⟩ ASSIGNS EXPRESSION
| INCDEC ⟨*identifier*⟩
| ⟨*identifier*⟩ INCDEC

ARITHMETICMORE → , ARITHMETIC
| ε

ASSIGNS → = | += | -= | *= | /=

EQUALITY → > | >= | < | <= | == | !=

ADDITIVES → + | -

MULTIPLICATIONS → * | /

INCDEC → `++` | `--`

EXPRESSION → EXPRESSION EQUALITY ADD
| ADD

ADD → ADD ADITIVES MULTIPLE
| MULTIPLE

MULTIPLE → MULTIPLE MULTIPLICATIONS IDS
| IDS

IDS → `(` EXPRESSION `)`
| ⟨*constant*⟩
| ⟨*identifier*⟩
| `call` ⟨*identifier*⟩ `(` VARIABLECALL `)`
| – IDS
| INCDEC IDS
| IDS INCDEC
| ε

}

D Tabulka podmínek a intervalových výsledků

Tato tabulka udává výsledky intervalů, které jsou větší, menší, větší rovno, menší rovno, rovny, či nerovny, než 5 obecných skupin intervalů ($(-\infty, \texttt{max})$, $(\texttt{min}, \texttt{max})$, $(\texttt{min}, +\infty)$, *Top*, *Bottom*). V těchto podmínkách může být ještě použito unární mínus, proto je třeba zkoumat jak kladnou, tak zápornou verzi.

Operace	**znaménko**	$(-\infty, \texttt{max})$	$(\texttt{min}, \texttt{max})$	$(\texttt{min}, +\infty)$	*Top*	*Bottom*
$>$	$+$	$\top$	$(\texttt{min}{+}1, +\infty)$	$(\texttt{min}{+}1, +\infty)$	$\top$	$\bot$
$>$	$-$	$(\texttt{max}{*}(-1)+1, +\infty)$	$(\texttt{max}{*}(-1)+1, +\infty)$	$\top$	$\top$	$\bot$
$\geq$	$+$	$\top$	$(\texttt{min}, +\infty)$	$(\texttt{min}, +\infty)$	$\top$	$\bot$
$\geq$	$-$	$(\texttt{max}{*}(-1), +\infty)$	$(\texttt{max}{*}(-1), +\infty)$	$\top$	$\top$	$\bot$
$<$	$+$	$(-\infty, \texttt{max}{-}1)$	$(-\infty, \texttt{max}{-}1)$	$\top$	$\top$	$\bot$
$<$	$-$	$\top$	$(-\infty, \texttt{min}{*}(-1)-1)$	$(-\infty, \texttt{min}{*}(-1)-1)$	$\top$	$\bot$
$\leq$	$+$	$(-\infty, \texttt{max})$	$(-\infty, \texttt{max})$	$\texttt{Top}$	$\top$	$\bot$
$\leq$	$-$	$\texttt{Top}$	$(-\infty, \texttt{min}{*}(-1))$	$(-\infty, \texttt{min}{*}(-1))$	$\top$	$\bot$
$=$	$+$	$(-\infty, \texttt{max})$	$(\texttt{min}, \texttt{max})$	$(\texttt{min}, +\infty)$	$\top$	$\bot$
$=$	$-$	$(\texttt{max}{*}(-1), +\infty)$	$(\texttt{max}{*}(-1), \texttt{min}{*}(-1))$	$(-\infty, \texttt{min}{*}(-1))$	$\top$	$\bot$
$\neq$	$+$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\neq$	$-$	$\top$	$\top$	$\top$	$\top$	$\bot$

E Složitější kód určen k testování programu

Tento kód nemá žádné praktické využití, je navržen tak aby otestoval zástupce analýz abstraktní interpretace a obsahoval větvení pro znázornění v CFG.

```
function void funkce1 (int a, int b)
{
  int u;
  int c = 20;
  int h = a;
  if (c > h)
  {
    b = 5;
  }
  else
  {
    b = 20;
  }
  a = 5;
  for (int i = -5;i > -10; i-)
  {
    u = 15;
    if(u < 10)
    {
      a = 15;
    }
  }
}
```

F Příloha na CD/DVD

Výsledkem bakalářské práce je program, který je uveden jako příloha na disku. Obsah tohoto disku je složka KerLang.

V této složce, jsou veškeré materiály potřebné ke správnému spuštění programu.

Cesta ke spustitelnému souboru je následující — "KerLang\bin\Release\KerLang.exe".

Složka "KerLang\Output"obsahuje ukázkové výsledky programu.

Další významy souborů a jejich použití, je vysvětleno v Kapitole 6 a také v Kapitole 2.

Všechny soubory, které ovlivňují práci programu, jsou nastaveny tak, aby byl program spustitelný.